Remarks

Reconsideration of the application is respectfully requested in view of the following remarks. Claims 1, 3-7, 9, 10, 12-24, and 26-32 are pending in the application. No claims have been allowed. Independent claim 14 and dependent claims 28 and 29 are currently amended. Claims 1, 6, 12, 14, and 24 are independent.

Interview

Applicants thank the Examiner for his time during the telephone interview on Nov. 18, 2008. Applicants discussed the difference between the unknown type of claim 1 compared to the iUnknown interface pointer described in IL Assembler. No agreement was reached.

Cited Art

The Office action ("Action") applies the following cited art: U.S. Patent No. 6,560,774 to Gordon (*Gordon*); and Lidin, *Inside Microsoft .NET IL Assembler*, (*Lidin*).

§ 101 Rejection

The Action rejects claims 14-15, 17-23, and 32 under 35 U.S.C. § 101 as being directed to non-statutory subject matter. The Applicants disagree, but in the interest of expediting patent prosecution have amended claim 14 to read:

> 14. A computer system for type-checking an intermediate representation of source code in a compiler comprising:
>
> a computer-readable storage medium containing one or more types associated with elements of the intermediate representation, wherein at least one of the types, designated as an unknown type, indicates an element can be of any type;
>
> a computer-readable storage medium containing one or more rule sets comprising rules associated with the type, designated as the unknown type, indicating an element can be of any type; and
>
> a type-checker module, wherein the type-checker is configured for applying the one or more rule sets to the elements of the intermediate representation.

Support for this amendment may be found in the original application as filed at, for example, page 16, lines 13-27. Dependent claims 15, 17-23, and 32 depend from claim 14. Applicants submit that amended claims 14-15, 17-23, and 32 are in condition for allowance, and such action is respectfully requested.

The Action rejects claims 28-29 under 35 U.S.C. § 101 as being directed to a computer readable medium. The Applicants disagree, but in the interest of expediting patent prosecution have amended claims 28 and 29 to recite: "A computer-readable storage medium containing computer-executable instructions . . . ." Support for these amendments may be found in the original application as filed at, for example, page 16, lines 23-27. Applicants submit that amended claims 28 and 29 are in condition for allowance, and such action is respectfully requested.

§ 103 Rejection

The Action rejects claims 1, 3-7, 9-10, 12-24, and 26-32 under 35 U.S.C. § 103(a) as being unpatentable over *Gordon* in view of *Lidin*. The Examiner's rejections are respectfully traversed.

Claim 1

***Gordon* does not disclose translating a code segment to an intermediate language, wherein one of the source languages is *an untyped source language*.**

Claim 1 recites:

> A method of type-checking a code segment written in a programming language comprising:
>
> translating the code segment from the programming language to one or more representations of an intermediate language, wherein the one or more representations of the intermediate language are capable of representing programs written in a plurality of different source languages, wherein the plurality of different source languages comprise at least one typed source language and at least one untyped source language; and
>
> type-checking the one or more representations based on a rule set, wherein the rule set comprises rules for type-checking a type designated as an unknown type, wherein the unknown type indicates that an element of the representation is of a type that is not known.

The Examiner rejects claim 1 as being unpatentable over *Gordon* and *Lidin*, contending that *Gordon* describes a method of type-checking, including "at least one untyped source language (e.g., SmallTalk, col. 35: 8-20)" (Action, p. 5, line 4). The Examiner's rejection is respectfully traversed. *Gordon* does not describe untyped languages; what *Gordon* describes are *dynamically typed languages*:

> By-ref parameters and value classes are sufficient to support statically typed languages (Java, C++, Pascal, etc.). They also support dynamically typed languages that pay a performance penalty to box value classes before passing them to polymorphic methods (Lisp, Scheme, SmallTalk, etc.).

(*Gordon*, col. 8 at). In fact, *Gordon* does not mention untyped languages at all. This deficiency is not cured by *Lidin*. For at least this reason, *Gordon* does not teach or suggest, alone or in combination with *Lidin*, the limitation of claim 1, "wherein the plurality of different source languages comprise at least one typed source language and at least one untyped source language." The Applicants would also like to respectfully point out that this rejection was already addressed by the Applicants in the Office Action Response of May 16, 2008 at page 10.

***Lidin* does not describe unknown types as in claim 1.**

The Examiner rejects claim 1 as being obvious over *Gordon* in view of *Lidin*, stating that "*Lidin* further discloses a type designated as an unknown type, wherein the unknown type indicates that an element of the representation is of a type that is not known," citing pages 10 and pages 12 of the reference provided by the Examiner:[1] "page 10, an unknown native type as IUNKNOWN; page 12, an unknown variant type as VT_UNKNOWN." (Action, p. 5, lines 15-18). The Examiner's rejection is respectfully traversed.

Lidin does not describe the limitation of claim 1, a method for type checking including a rule set, including rules for type-checking "a type designated as an unknown type, wherein the unknown type indicates that an element of the representation of a type is of a type that is not known." Instead of an unknown type, what *Lidin* page 166 discloses is the .NET Framework *interface* type named "IUnknown," which is described as an "IUnknown interface pointer." This unmanaged type is merely a pointer to the interface of a Microsoft COM managed object. It is treated as a native type defined in the common language runtime:

[1] In the Action, Examiner cited "Inside Microsoft .NET *Lidin*," ISBN 0-7356-1547-0, which does not have page numbers. Applicants have provided a copy of portions of a hardcopy of the same book, filed concurrently with an Information Disclosure Statement herewith, and cited to the hardcopy page numbers.

> When managed code calls unmanaged methods or exposes managed fields to unmanaged code, it is sometimes necessary to provide specific information about how the managed types should be marshaled to and from the unmanaged types. The unmanaged types recognizable by the common language runtime are referred to as *native*[.]

(*Lidin*, p. 165). *Lidin* then lists native types in table 7-6, including integers, floats, and "IUnknown interface pointer" and "IDispatch interface pointer." *Id.* at 166. *Lidin* goes on to discuss variant types, which "are defined in the enumeration VARENUM in the Wtypes.h file." (*Lidin* at 168). *Lidin* lists variant types in Table 7-7, including integers, floats, VT_UNKNOWN, which refers to the iUnknown type, and the VT_DISPATCH, which refers to the iDispatch type. *Id.* While these tables do not discuss in detail the iUnknown type, *Lidin* does further describe the use of the iUnknown interface pointer in chapter 15. There, *Lidin* describes COM callable wrappers (CCWs) for managed objects:

> [T]he runtime creates a COM callable wrapper [CCW], which serves as a proxy for the object. Because a CCW is not subject to the GC [garbage collection] mechanism, it can be referenced from unmanaged code without causing any ill effects.

Lidin at 354. *Lidin* further describes that its runtime environment maintains the same IUnknown *interface* for instances of the same object:

> The runtime carefully maintains a one-to-one relationship between a managed object and its CCW, not allowing an alternative CCW to be created. This guarantees that all interfaces of the same object relate to the same *IUnknown* and that the interface queries are consistent.

Id. at 355. *Lidin* further describes that IUnknown and IDispatch are automatically implemented interfaces for the CCWs generated by the common language runtime:

> A CCW generated by the common language runtime for each instance of the exposed managed class also implements other interfaces not explicitly implemented by the class. In particular, a CCW automatically implements *IUnknown* and *IDispatch*.

Id. at 363. Finally, *Lidin* specifically distinguishes between what it calls "real" types and a special kind of type, interface types (such as IUnknown and IDispatch):

> An interface is a special kind of type, defined . . . as "a named group of methods, locations, and other contracts that shall be implemented by any object type that supports the interface contract of the same name." In other words, *an interface is not a "real" type but merely a named descriptor of methods and*

> *properties exposed by other types.* Conceptually, an interface in the common language runtime is similar to a COM interface . . . . [n]ot being a real type, an interface is not derived from any other type, nor can other types be derived from an interface."

Id. at 140–41 (emphasis added).

Therefore, one skilled in the art would not interpret the inclusion of iUnknown, the well-known Microsoft COM object *interface function type* in a list of types, to teach or suggest an *unknown data type* (*i.e.* as indicating "that an element of the representation is of a type that is now known") as in claim 1. It is proper to consider the material in *Lidin* at pages 355–363 in addition to those cited by the Examiner in ascertaining the meaning of the "iUnknown" interface pointer: "[a] prior art reference must be considered in its entirety, i.e., as a whole, including portions that would lead away from the claimed invention." MPEP § 2141.02, part VI.

The Examiner further contends that the iUnknown interface describes an unknown type. (Examiner's Interview Summary of Nov. 21, 2008, at p. 4). Specifically, the Examiner contends that "QueryInterface, at runtime, must receive a required input parameter 'iid' to identify the type of the interface requested." (Interview Summary, p. 4). The Examiner cites Exhibit B, attached herewith. Exhibit B describes the method IUnknown::QueryInterface, which returns a pointer to a the interface specified by the input parameter iid. (Exhibit B, p. 1).

Applicants submit that IID does not identify the *type* of the interface, but the globally unique *name* of the interface. In support, Applicants cite to Richard Grimes, *Professional DCOM Programming*, Chapter 3 (Wrox Press 1997) (*Grimes*), which Applicants are citing in an Information Disclosure Statement filed concurrently. Specifically, page 103 of Grimes defines the IID as the globally unique name of the interface:

> "We have established that an object can specify whether it supports a requested interface. However, there must be some mechanism to identify the interface that we are interested in. This is done with an IID or Interface ID. IIDs are just GUIDs used to identify interfaces. GUID stands for Globally Unique Identifier and each GUID is a 128-bit number."

(*Grimes*, p. 103). "[Y]ou can ask an activated object if it supports a particular interface by passing the IID to the QueryInterface() of the object." (*Id.*) Further, QueryInterface returns "a pointer to the interface in the pointer referenced by ppvObject," not the *type* of the interface, and a return value of either S_OK or E_NOINTERFACE, depending on whether the object implements the requested interface. (*Grimes*, p. 101). Nothing in *Grimes* or Exhibit B mentions

"interface *type*" or "to identify the *type* of the interface requested," as contended by the Examiner. (Interview Summary, p. 4) (emphasis added). Other methods related to iUnknown (*e.g.*, AddRef, Release) do not describe type checking or unknown types.

For at least the foregoing reasons, *Gordon* does not teach or suggest alone, or in combination with any prior art of record, the method of claim 1, which is in condition for allowance, and such action is respectfully requested.

Claim 6

The Examiner rejects claim 6 over *Gordon* in view of *Lidin*. The Examiner's rejection is respectfully traversed.

***Gordon* in view of *Lidin* does not describe determining whether to retain type information using a rule set comprising types designated as unknown types.**

The Examiner contends that *Gordon* describe for each intermediate language representation "determining whether to retain type information for one or more elements of the representation," and "wherein the rule set comprises rules for type-checking the type designated as the [un]known type, citing *Gordon*, FIG. 23; column 27, lines 4–33; column 17 line 52 through column 18 line 28; column 20 lines 44–57; column 27 line through column 28 line 29. Because Gordon in view of Lidin does not disclose each and every limitation, claim 6 is allowable.

FIG. 23 and the cited text describe what *Gordon* calls a "Virtual Execution System" (VES). (*Gordon*, col. 3, lines 7–8; col. 27, lines 12–13). *Gordon* recites that a VES can skip the step that *Gordon* calls "verification" for trusted code, and that the VES can also skip verification of precompiled native code that is fully trusted. (*Gordon*, col. 27, lines 12–33). "Verification" apparently includes checking IL code metadata and code for consistency and accuracy, including semantic checks such as "reference aspect checks (such as byref and refany checks), value class checks, native-size primitive type checks, and tail call verifications." (*Gordon*, col. 6, line 58 through col. 7, line 5). *Gordon* teaches that the VES stops maintaining type information to aid verification: "the type of the arguments to any method in the code being verified *are fixed*. . . . This means that dynamic type information for arguments *do not need to be maintained* on a per-basic block basis." (*Gordon*, col. 7, lines 57–60) (emphasis added). *Gordon* further teaches that type information for primitive locals is "fixed" and that for non-primitives, type information must be maintained:

> [T]he type of primitive local variables, such as integers, floating points, etc., *is fixed*. If a variable is declared as being a primitive type, then it is not allowed to store a non-primitive type, such as an object reference, into it. In this manner, a single bit--"live" or "dead" for example--is sufficient to convey a primitive variable's state at any point. For non-primitive variables, however, *complete type information must be maintained*.

(*Gordon*, col. 7, line 66 through col. 8, line 6). Gordon, therefore, discloses a method of verification that either does not maintain type information for arguments, maintains complete type information as designated in the source code. This is not the same method as claim 6, which, after determining whether to retain type information for elements of the intermediate language representation, designates one or more elements "as an unknown type." Applying the method of claim 6, the elements designated as an unknown type are type checked using a rule set that includes "rules for type-checking the type designated as the unknown type."

***Gordon* in view of *Lidin* does not describe designating unknown types.**

The Examiner also contends that *Gordon* discloses, *inter alia*, "associating one or more elements of the representation with a type, designated as an [un]known type, indicating the element can be of any type; and type-checking the one or more representations based on a rule set. . . ." (Action, p. 6) (citations omitted). What the cited portions of *Gordon* describe are the "Supported Data Types: Natural Size: I, R, U, O, and &" that apparently include natural size: two's complement signed value, unsigned binary value/unmanaged pointer, floating point value, object reference to managed memory/managed pointer, and pointers to the interior of an object. (*Gordon*, FIG. 24; col. 27, lines 53–65; col. 28, lines 39–40). These types "are a mechanism in the EE [Execution Engine] for deferring the choice of a value's size." (*Gordon*, col. 27, lines 64–65). Further, *Gordon* recites that unmanaged pointers are desirable in order "to defer the choice of pointer size from compile time to EE initialization time." (*Gordon*, col. 21–24). Therefore, *Gordon* merely describes a limited set of "Supported Data Types," including an unmanaged pointer type, which allows deferring the choice of pointer size until Execution Engine initialization time. Hence, *Gordon* does not teach or suggest, and in fact teaches away from, the *unknown* types of the method of claim six, which includes type-checking representations of an intermediate language, included elements "designated as an unknown type, indicating the element can be of any type." This deficiency is not cured by *Lidin*, as already

discussed in the remarks regarding claim 1. Therefore, claim 6 is in condition for allowance, and such action is respectfully requested.

Request for Interview

If any issues remain, the Examiner is formally requested to contact the undersigned attorney prior to issuance of the next Office action in order to arrange a telephonic interview. It is believed that a brief discussion of the merits of the present application may expedite prosecution. Applicants submit the foregoing remarks so that the Examiner may fully evaluate Applicants' position, thereby enabling the interview to be more focused.

This request is being submitted under MPEP § 713.01, which indicates that an interview may be arranged in advance by a written request.

Conclusion

The claims should be allowable. Such action is respectfully requested.

Respectfully submitted,

KLARQUIST SPARKMAN, LLP

One World Trade Center, Suite 1600
121 S.W. Salmon Street
Portland, Oregon 97204
Telephone: (503) 595-5300
Facsimile: (503) 595-5301

By /Cory A. Jones/
Cory A. Jones
Registration No. 55,307